9 Python Object Oriented Programming (OOP)

0.1 Objective

The objectives of the experiment is to learn the following:

- Give a quick introduction about Python object oriented Programming (OOP).
- Define classes in Python.
- Show some examples about Inheritance.
- Explain what are Polymorphism in OOP.
- Show Difference between method Overloading and Overriding.
- Show how to handle any unexpected error in your Python programs.

0.2 Introduction to Python OOP

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This experiment helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented OO programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts. However, here is small introduction of Object-Oriented Programming OOP to bring you at speed.

Class: A class describes all the attributes of objects, as well as the methods that implement the behavior of member objects. It is a comprehensive data type, which represents a blue print of objects. It is a template of object.

Object: Object is instance of classes. It is a basic unit of a system. An object is an entity that has attributes, behavior, and identity. Attributes and behavior of an object are defined by the class definition**.**

Polymorphism: using an entity in multiple forms.

Inheritance: Promotes the reusability of code and eliminates the use of redundant code. It is the property through which a child class obtains all the features defined in its parent class. When a class inherits the common properties of another class, the class inheriting the properties is called a derived class and the class that allows inheritance of its common properties is called a base class.

Function Overriding: Overriding involves the creation of two or more methods with the same name and same signature in different classes (one of them should be parent class and other should be child).

Function Overloading: Overloading is a concept of using a method at different places with same name and different signatures within the same class.

Abstraction: Refers to the process of exposing only the relevant and essential data to the users without showing unnecessary information.

0.3 Classes in Python

Create Class In python

The ***class*** statement creates a new class definition. The name of the class immediately follows the keyword ***class*** followed by a colon as follows:

```
class ClassName:
  'Optional class documentation string'
  class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

Example1: open **employee**.py file

```
class Employee:
  'Common base class for all employees'
  empCount = 0
  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1
  def displayCount(self):
   print("Total Employee %d" % Employee.empCount)
  def displayEmployee(self):
    print("Name : ", self.name,  ", Salary: ", self.salary)
```

- The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the ***self*** argument to the list for you; you do not need to include it when you call the methods.

Getters and Setters

Getters and setters are used in many object oriented programming languages to ensure the principle of data encapsulation.
The following example demonstrate how we can design a class with getters and setters to encapsulate the private attribute "self.__x":

```
class P:
    def __init__(self,x):
        self.__x = x
    def get_x(self):
        return self.__x
    def set_x(self, x):
        self.__x = x
```

Creating Instance Object

To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.

```
from employee import *
#This would create first object of Employee class
emp1 = Employee("Aziz", 15000)
#This would create second object of Employee class
emp2 = Employee("Mohammad", 1000)
```

Accessing Attributes

You access the object's attributes using the **dot** operator with object. Class variable would be accessed using class name as follows :

```
emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

You can delete a single object or multiple objects by using the **del** statement.

```
del var
del var_a, var_b
```

You can add, remove, or modify attributes of classes and objects at any time

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions –

- The **getattr(obj, name[, default])** : to access the attribute of object.
- The **hasattr(obj,name)** : to check if an attribute exists or not.
- The **setattr(obj,name,value)** : to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** : to delete an attribute.

```
print(hasattr(emp1, 'age'))    # Returns true if 'age' attribute exists
print(getattr(emp1, 'name'))    # Returns value of 'age' attribute
setattr(emp1, 'salary', 8) # Set attribute 'age' at 8
delattr(emp1, 'salary')    # Delete attribute 'age'
```

0.4 Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute :

- **__dict__:** Dictionary containing the class's namespace.
- **__doc__:** Class documentation string or none, if undefined.
- **__name__:** Class name.
- **__module__:** Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__:** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes:

Example2:

```
#!/usr/bin/python

class Employee:
  'Common base class for all employees'
  empCount = 0
  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1
  def displayCount(self):
   print("Total Employee %d" % Employee.empCount)
  def displayEmployee(self):
    print("Name : ", self.name,  ", Salary: ", self.salary)
print("Employee.__doc__:", Employee.__doc__)
print("Employee.__name__:", Employee.__name__)
print("Employee.__module__:", Employee.__module__)
print("Employee.__bases__:", Employee.__bases__)
print("Employee.__dict__:", Employee.__dict__)
```

Output:

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

0.5 Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name :

```
class SubClassName ([ParentClass1, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

Example3:

```
#!/usr/bin/python
class Parent:       # define parent class
  parentAttr = 100
  def __init__(self):
    print("Calling parent constructor")
  def parentMethod(self):
    print('Calling parent method')
  def setAttr(self, attr):
    Parent.parentAttr = attr
  def getAttr(self):
    print("Parent attribute :", Parent.parentAttr)
class Child(Parent): # define child class
  def __init__(self):
    print("Calling child constructor")
  def childMethod(self):
    print('Calling child method')
c = Child()         # instance of child
c.childMethod()     # child calls its method
c.parentMethod()    # calls parent's method
c.setAttr(200)      # again call parent's method
c.getAttr()         # again call parent's method
```

Output:

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows :

```
class A:        # define your class A
.....
class B:        # define your class B
.....
class C(A, B):   # subclass of A and B
.....
```

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class.

0.6 Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example4:

```
#!/usr/bin/python
class Parent:                    # define parent class
  def myMethod(self):
    print('Calling parent method')
class Child(Parent):             # define child class
  def myMethod(self):
    print('Calling child method')
c = Child()                      # instance of child
c.myMethod()                     # child calls overridden method
```

0.7 Overloading Methods

Following table lists some generic functionality that you can override in your own classes –

SN	Method, Description & Sample Call
1	**__init__ (self [,args...])** Constructor (with any optional arguments) Sample Call : *obj = className(args)*
2	**__del__(self)** Destructor, deletes an object Sample Call : *del obj*
3	**__str__(self)** Printable string representation Sample Call : *str(obj)*
4	**__cmp__ (self, x)** Object comparison Sample Call : *cmp(obj, x)*

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation.

Example5:

```
#!/usr/bin/python
class Vector:
  def __init__(self, a, b):
    self.a = a
    self.b = b
  def __str__(self):
    return 'Vector (%d, %d)' % (self.a, self.b)
  def __add__(self,other):
    return Vector(self.a + other.a, self.b + other.b)
v1 = Vector(2,10)
v2 = Vector(5,-2)
print(v1 + v2)
```

Output:

```
Vector(7,8)
```

0.8 Attributes types and Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

The following table summarize the attribute types:

Naming	Type	Meaning
name	Public	These attributes can be freely used inside or outside of a class definition.
_name	Protected	Protected attributes should not be used outside of the class definition, unless inside of a subclass definition.
__name	Private	This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside of the class definition itself.

Example6:

```
#!/usr/bin/python
class JustCounter:
  __secretCount = 0

  def count(self):
    self.__secretCount += 1
    print(self.__secretCount)
counter = JustCounter()
```

```
counter.count()
counter.count()
print(counter.__secretCount)
```

Output:

```
1
2
Traceback (most recent call last):
 File "test.py", line 10, in <module>
  print(counter.__secretCount)
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className__attrName*. If you would replace your last line as following, then it works for you –

```
.....................
print(counter._JustCounter__secretCount)
```

When the above code is executed, it produces the following result –

```
1
2
2
```

0.9 Polymorphism:

Sometimes an object comes in many types or forms. If we have a button, there are many different draw outputs (round button, check button, square button, button with image) but they do share the same logic: onClick(). We access them using the same method . This idea is called *Polymorphism*. Polymorphism is based on the greek words Poly (many) and morphism (forms). We will create a structure that can take or use many forms of objects.

Example7:

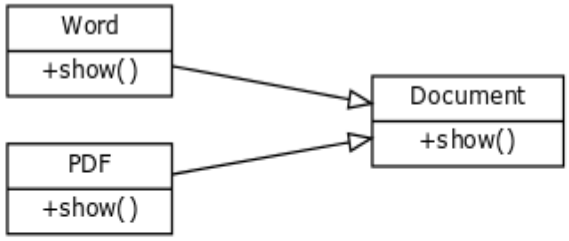
We create two classes: Bear and Dog, both can make a distinct sound. We then make two instances and call their action using the same method.

```
class Bear(object):
   def sound(self):
      print("Groarrr")
class Dog(object):
   def sound(self):
      print("Woof woof!")
def makeSound(animalType):
   animalType.sound()
bearObj = Bear()
dogObj = Dog()
makeSound(bearObj)
makeSound(dogObj)
```

Output:

```
Groarrr
Woof woof!
```

0.10 Polymorphism with abstract class (most commonly used)



Polymorphism visual.
Abstract structure is defined in Document class.

If you create an editor you may not know in advance what type of documents a user will open (pdf format or word format)?.
Wouldn't it be great to access them like this, instead of having 20 types for every document?

```
for document in documents:
    print(document.name + ': ' + document.show())
```

To do so, we create an abstract class called document. This class does not have any implementation but defines the structure (in form of functions) that all forms must have. If we define the function show() then both the PdfDocument and WordDocument must have the show() function. Full code:

Example8:

```
class Document:
    def __init__(self, name):
        self.name = name
    def show(self):
        raise NotImplementedError("Subclass must implement abstract method")
class Pdf(Document):
    def show(self):
        return 'Show pdf contents!'
class Word(Document):
    def show(self):
        return 'Show word contents!'

documents = [Pdf('Document1'),
             Pdf('Document2'),
             Word('Document3')]

for document in documents:
    print(document.name + ': ' + document.show())
```

Output:

```
Document1: Show pdf contents!
Document2: Show pdf contents!
Document3: Show word contents!
```

0.11 Python Exceptions Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them:

- **Exception Handling**
- **Assertions**

In this Experiment, Exception Handling would be covered.

Exception Handling

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

List of some Standard Exceptions :

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
IndexError	Raised when an index is not found in a sequence.
NameError	Raised when an identifier is not found in the local or global namespace.

Syntax

Here is simple syntax of ***try.... except ...else*** blocks:

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
```

```
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax −

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example9:

```
#!/usr/bin/python
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print("Error: can\'t find file or read data")
else:
   print("Written content in the file successfully")
   fh.close()
```

The *except* Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows −

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The *except* Clause with Multiple Exceptions

You can also use the same *except* statement to handle multiple exceptions as follows −

```
try:
   You do your operations here;
   ......................
except([Exception1, Exception2,...ExceptionN]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block
```

The try-finally Clause

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this:

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

You cannot use *else* clause as well along with a finally clause.

Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

Syntax

```
raise [Exception [, args [, traceback]]]
```

Example:

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows :

```
def functionName( level ):
   if level < 1:
      raise "Invalid level!", level
      # The code below to this would not be executed
      # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows:

```
try:
  Business Logic here...
except "Invalid level!":
  Exception handling here...
else:
  Rest of the code here...
```

0.12 todo

This part will be given to you by the teacher assistant in the lab time.